МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

Отчет
по лабораторной работе №2
по дисциплине «Алгоритмы и структуры данных»»
Тема: Иерархические списки

Студент гр. 8304 _____ Завражин Д.Г.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург

2019

Цель работы.

Ознакомиться с основными понятиями и приёмами программной реализации иерархических списков, освоить навыки разработки и написания процедур их обработки на языке C++ на примере поставленного задания.

Задание.

Вариант 19

Пусть *арифметическое* выражение представлено иерархическим списком. В выражение входят константы и переменные, которые являются атомами списка. Операции представляются в *постфиксной* форме (<аргументы> <операция>)). Аргументов может быть 1, 2 и более. Например (в префиксной форме): (+ a (* b (- c))).

В задании даётся следующий вариант требуемого действия с выражением: *проверка синтаксической корректности и деления на 0*.

В индивидуальном задании указывается: тип выражения (возможно дополнительно - состав операций), вариант действия и форма записи. Указания, соответствующие данному варианту, подставлены в текст задания и выделены курсивом.

Класс *TrivariateHierarchicalList*

С целью программной реализации структуры данных иерархического списка на основе указателей была создана структура данных *TrivariateHierarchicalList*, представляющая собой иерархический список, каждый элемент которого хранит либо указатель на подсписок, либо одит из задаваемых шаблоном типов данных.

Интерфейс класса *TrivariateHierarchicalList* состоит из:

- Конструктора, инициализирующего первый элемент списка;

- Деструктора, используемого для освобождения памяти;
- Метода represent, возвращающего строковое представление иерархического списка;
- Метода begin, возвращающего итератор начала списка;
- Метода end, возвращающего итератор конца списка.

Они имеют слеующие сигнатуры:

- TrivariateHierarchicalList()
- TrivariateHierarchicalList()
- std::string represent()
- Iterator begin()
- Iterator end()

Класс *TrivariateHierarchicalList* имеет два недоступных извне вложенных класса: *Node* и *Iterator*.

Реализация класса *TrivariateHierarchicalList* и вложенных в него классов, находящаяся в заголовочном файле *trivariatehierarchicallist.h*, приведена вместе со всем исходным кодом программы в приложении А.

Класс *Node*

Класс *Node* реализует один элемент иерархического списка и написан на основе типа данных std::variant из стандартной библиотеки языка C++17. Его интерфейс состоит из:

- Конструктора;

- Деструктора, используемого для освобождения памяти;
- Геттера next и сеттера setNext, обеспечивающих доступ к указателю, указывающему на следующий элемент списка;
- Геттера content и сеттера setContent, обеспечивающих доступ к содерэимому элемента списка;
- Метода represent, возвращающего строковое представление иерархического списка с данным элементом в качестве головного.

Сигнатуры методов класса *Node*:

- Node()
- void setNext(Node* const next)
- const std::variant<T, U, V, Node*>& content()
- void setContent(const T& content)
- void setContent(const U& content)
- void setContent(const V& content)
- void setContent(Node* const content)
- std::string represent()

Класс *Iterator*

Класс *Iterator* реализует нерекурсивную итерацию по атомарным элементам иерархического списка. Его интерфейс состоит из:

- Конструктора, запоминающего указатель, указывающий на хранимый в нём элемент и переходящий к первому атомарному после него;

- Унарных операторов инкремента;
- Бинарных операторов равенства и неравенства;
- Метода getPreviousNodeCount, возвращающего количество элементов на том же уровне до него.

Заметим, что освобождение памяти по указателю, хранящемуся у данного итератора, не требуется, так как эта память будет освобождена при вызове деструктора класса *TrivariateHierarchicalList*.

У него также имеется недоступный извне метод normalizePosition, реализующий переход к следующему атомарному элементу.

Методы класса *Iterator* имеют следующие сигнатуры:

- Iterator(Node *node=nullptr)
- Iterator& operator++()
- Iterator operator++(int)
- Node& operator* ()
- Node* operator-> ()
- bool operator== (const Iterator& that)
- bool operator!= (const Iterator& that)
- size_t getPreviousNodeCount()
- void normalizePosition()

Класс *Expression*

С целью программной реализации структуры данных для представления строки данного в условии задания вида на основе иерархического списка была создана структура данных *Expression*, представляющая собой подкласс иерархического списка с добавлением необходимыхдля работы с выражениями методов.

Интерфейс класса *Expression*, вдобавок к интерфейсу класса *TrivariateHierarchicalList*, состоит из:

- Конструктора, принимающего строку;
- Метода isCorrect, проверяющего корректность выражения;
- Метода getErrors, возвращающего список ошибок в выражении.

Вдобавок к этому, он имеет следующие недоступныые извне методы:

- Рекурсивного метода parse, преобразующего переданную конструктором строку в иерархический спмсок;
- Итеративного метода checkNodes, проводящего проверку вычислимости выражения и деления на 0 путём попытки вычисления выражения на стеке без подстановки неизвестных переменных.

Методы класса *Expression* имеют следующие сигнатуры:

- Expression(const std::string& expression)
- bool isCorrect()
- std::string getErrors()
- std::string::const_iterator parse(ExpressionNode *node, std::string::const_iterator current, std::string::const_iterator end)

- void checkNodes(std::stack<std::variant<T,bool»& executionStack)

Ошибки, обнаруженные в некотором выражении, хранятся в классе в двух векторах строк, parsingErrors и executionErrors, соответственно хранящие ошибко, обнаруженные при выполнении методов parse и checkNodes.

Реализация класса *Expression*, находящаяся в заголовочном файле *expression.h*, приведена вместе со всем исходным кодом программы в приложении А.

Метод *parse*

Метод *parse* класса *Expression* выполняет рекурсивное преобразование строки типа std::string, чьи итераторы переданы ему, в иерархический список, представленный данным классом; при этом метод вызывается только для каждого уровня списка.

Метод *checkNodes*

Метод *checkNodes* класса *Expression* выполняет итеративную проверку выполнимости хранящегося в классе выражения. При этом проводятся проверки на:

- Соответствие количества операндов операции;
- Постфиксность выражения;
- Деление на 0.

Используемый алгоритм для избежпния рекурсии прибегает к описанному ранее итератору *TrivariateHierarchicalList::Iterator* и непосредственным действиям со стеком. В процессе вычислений промежуточные значения хранятся как std::variant<T,bool> >, где T — тип числовых данных, а в типе данных bool хранятся недоступные для вычисления данные, которые могут быть:

- Значениями переменных;
- Результатами вычисления содержащих ошибки операций.

Вспомогательные функции и типы данных

В процессе выполнения работы были также созданы:

- Класс перечисления lab2::OperationType, для хранения кодов операции;
- Фунция lab2::to_string, эквивалентная функции std::to_string, перегруженной для типов данных std::string и класса перечисления lab2::OperationType;
- Функция lab2::stoT, приводящая строковый тип данных std::string к заданному целочисленному формату.

Их реализация, находящаяся в заголовочном файле utils.h, приведена вместе со всем исходным кодом программы в приложении А.

Функция *main*

Функция *main* выполняет задачу получения от пользователя строки, содержащей анализируемое выражение. Это может происходить двумя способами:

- Посредством передачи в качестве единственного аргумента командной строки;
- Посредством ввода по прямому запросу программы.

После получения пути программа передаёт его функции конструктору класса Expression, выполняющему вызов его метода parse, обеспечивающего преобразование строки в икрархический список.

Сигнатура функции main: int main(int argc, char* argv[]).

Тестирование программы

Тесты, содержащиеся в файле *tests.txt* и важные с точки зрения оценки работ программы фрагменты её вывода, приведены в таблице 1.

Таблица 1 – Тесты, применённые при тестирование программы.

№	Тест	Обнаруженные ошибки
1		The provided expression is empty
2	()	The provided expression contains empty parentheses
2	(3 a)	There is 1 unused operand left.
2	(5 6 +)	The string contains the following character(s) after the last closing bracket: ”)”
2	(5 6 +) 5	The string contains the following character(s) after the last closing bracket: ” 5”
2	((5 6 +) 5)	There is 1 unused operands left.
2	(5 0 /)	Term 3: Division by 0 encountered.
2	(5 (5 5 -) /)	Term 5: Division by 0 encountered.
2	((6 (5 9 *) /) (4 (6 5/) -) +)	The given expression is correct.
2	((6 (5 9 *) /) (4 (6 0/) -) +)	Term 5: Division by 0 encountered.
2	((x1 (x2 x3 *) /)(x4 (x2 (5.5 x3*) /) -) +)	The given expression is correct.

На всех приведённых выше входных данных программа выдаёт ожидаемый результат; отсюда можно сделать вывод, что данная программа корректно

работает во всех охватываемых составленными тестами случаях.

Вывод

В результате выполнания лабораторной работы была реализована программа, отвечающая всем поставленным условиям и проходящая рассмотренное выше составленное в процессе выполнения работы тестирование. Помимо этого, были на практическом примере отточены навыки проектирования, написания и тестирования иерархических списков и алгоритмов работы с ними, владения языком C++.

Приложение А

Исходный код программы

Программа, использованная при выполнении лабораторной работы и представленная ниже, написана на языке программирования C++.

utils.h

```
#ifndef LAB2_UTILS_H_
#define LAB2_UTILS_H_

#include <string>

namespace lab2
{
    // enum class OperationType is used to encode an arithmetic
       operation
    enum class OperationType {ADDITION, SUBTRACTION, MULTIPLICATION
       , DIVISION};

    // lab2::to_string is similar to std::to_string, but is
       overloaded for two
    // additional types
    std::string to_string(const std::string& string)
    {
        return string;
    }

    std::string to_string(lab2::OperationType operationType)
    {
```

```
20          if(operationType == lab2::OperationType::ADDITION)
21              return "+";
22          if(operationType == lab2::OperationType::SUBTRACTION)
23              return "-";
24          if(operationType == lab2::OperationType::MULTIPLICATION)
25              return "*";
26          if(operationType == lab2::OperationType::DIVISION)
27              return "/";
28          return "?";
29      }
30
31      std::string to_string(auto value)
32      {
33          return std::to_string(value);
34      }
35
36      // lab2::stoT is a generalization of std::stoi, std::stol, ...
37      // for a numeric type T
38      template<class T>
39      T stoT(std::string);
40      template<>
41      short stoT<short>(std::string str){return std::stoi(str);}
42      template<>
43      int stoT<int>(std::string str){return std::stoi(str);}
44      template<>
45      long stoT<long>(std::string str){return std::stol(str);}
46      template<>
47      long long stoT<long long>(std::string str){return std::stoll(
            str);}
48      template<>
49      unsigned short stoT<unsigned short>(std::string str){return std
            ::stoul(str);}
50      template<>
```

```
51      unsigned int stoT<unsigned int>(std::string str){return std::
           stoul(str);}
52      template<>
53      unsigned long stoT<unsigned long>(std::string str){return std::
           stoul(str);}
54      template<>
55      unsigned long long stoT<unsigned long long>(std::string str){
           return std::stoull(str);}
56      template<>
57      float stoT<float>(std::string str){return std::stof(str);}
58      template<>
59      double stoT<double>(std::string str){return std::stod(str);}
60      template<>
61      long double stoT<long double>(std::string str){return std::
           stold(str);}
62  }
63
64  #endif  // LAB2_UTILS_H_
```

trivariatehierarchicallist.h

```
1   #ifndef LAB2_TRIVARIATEHIERARCHICALLIST_H_
2   #define LAB2_TRIVARIATEHIERARCHICALLIST_H_
3
4   #include <variant>
5   #include <stack>
6
7   #include "utils.h"
8
9   namespace lab2
10  {
```

```
11      // TrivariateHierarchicalList class was designed to store
           either a value of
12      // one out of three distinct types or a pointer to a sublist
13      template<class T, class U, class V>
14      class TrivariateHierarchicalList
15      {
16      protected:
17          class Node;
18      private:
19          class Iterator;
20
21      public:
22          explicit TrivariateHierarchicalList()
23          {
24              this->head_ = new Node();
25          }
26
27          ~TrivariateHierarchicalList()
28          {
29              delete this->head_;
30          }
31
32          std::string represent()
33          {
34              return this->head_->represent();
35          }
36
37          Iterator begin()
38          {
39              return Iterator(this->head_);
40          }
41
42          Iterator end()
```

```
43          {
44              return Iterator(nullptr);
45          }
46
47      protected:
48          Node* head()
49          {
50              return this->head_;
51          }
52
53      private:
54          Node* head_ = nullptr;
55
56      protected:
57          // The nested class Node was designed to be a single node
               of a
58          // trivariate hierarchical list
59          class Node
60          {
61          public:
62              explicit Node()
63              {
64                  this->content_ = nullptr;
65              }
66
67              ~Node()
68              {
69                  if(std::holds_alternative<Node*>(content_))
70                      delete std::get<Node*>(content_);
71                  delete this->next_;
72              }
73
74              Node* next()
```

```
75              {
76                  return this->next_;
77              }
78
79              void setNext(Node* const next)
80              {
81                  this->next_ = next;
82              }
83
84              const std::variant<T, U, V, Node*>& content()
85              {
86                  return this->content_;
87              }
88
89              void setContent(const T& content)
90              {
91                  this->content_ = content;
92              }
93
94              void setContent(const U& content)
95              {
96                  this->content_ = content;
97              }
98
99              void setContent(const V& content)
100             {
101                 this->content_ = content;
102             }
103
104             void setContent(Node* const content)
105             {
106                 this->content_ = content;
107             }
```

```
108
109             std::string represent()
110             {
111                 std::string representation = "(";
112                 auto current = this;
113                 while(current != nullptr)
114                 {
115                     if(std::holds_alternative<Node*>(current->
                           content_) &&
116                        std::get<Node*>(current->content()) !=
                              nullptr)
117                         representation +=
118                             std::get<Node*>(current->content_)->
                                   represent();
119                     else if(std::holds_alternative<T>(current->
                           content_))
120                         representation +=
121                             lab2::to_string(std::get<T>(current->
                                   content_));
122                     else if(std::holds_alternative<U>(current->
                           content_))
123                         representation +=
124                             lab2::to_string(std::get<U>(current->
                                   content_));
125                     else if(std::holds_alternative<V>(current->
                           content_))
126                         representation +=
127                             lab2::to_string(std::get<V>(current->
                                   content_));
128                     if(current->next_ != nullptr)
129                         representation += ' ';
130                     current = current->next_;
131                 }
```

```
132                 return representation + ')';
133             }
134 
135         private:
136             std::variant<T, U, V, Node*> content_;
137             Node *next_ = nullptr;
138         };
139 
140     private:
141         // The nested class Iterator was designed to facilitate
               iteration
142         // through a hierarchical list using explicit stack
               manipulations
143         class Iterator
144         {
145         public:
146             using difference_type = std::ptrdiff_t;
147             using value_type = Node;
148             using pointer = Node*;
149             using reference = Node&;
150             using iterator_category = std::forward_iterator_tag;
151 
152             explicit Iterator(Node *node)
153             {
154                 this->current = node;
155                 this->previousNodeCountStack.push(0);
156                 this->normalizePosition();
157             }
158 
159             Iterator& operator++()
160             {
161                 if(this->current == nullptr)
162                     return *this;
```

```
163                 this->current = this->current->next();
164                 previousNodeCountStack.top() += 1;
165                 this->normalizePosition();
166                 return *this;
167             }
168 
169             Iterator operator++(int)
170             {
171                 auto old = *this;
172                 ++(*this);
173                 return old;
174             }
175 
176             reference operator *()
177             {
178                 return *(this->current);
179             }
180 
181             pointer operator ->()
182             {
183                 return &**this;
184             }
185 
186             bool operator ==(const Iterator& that)
187             {
188                 return this->current == that.current &&
189                        this->nodeStack == that.nodeStack;
190             }
191 
192             bool operator !=(const Iterator& that)
193             {
194                 return this->current != that.current ||
195                        this->nodeStack != that.nodeStack;
```

```
196             }
197
198             size_t getPreviousNodeCount()
199             {
200                 return this->previousNodeCountStack.top();
201             }
202
203         private:
204             std::stack<Node*> nodeStack;
205             std::stack<size_t> previousNodeCountStack;
206             Node* current = nullptr;
207
208             // ensures that this->current holds a pointer to an
                   atomic node
209             void normalizePosition()
210             {
211                 while((this->current == nullptr && this->nodeStack.
                       size() > 0) ||
212                       (this->current != nullptr &&
213                       std::holds_alternative<Node*>(this->current->
                           content()) &&
214                       std::get<Node*>(this->current->content()) !=
                           nullptr))
215                 {
216                     while(this->current == nullptr && this->
                           nodeStack.size() > 0)
217                     {
218                         this->current = this->nodeStack.top()->next
                               ();
219                         this->nodeStack.pop();
220                         this->previousNodeCountStack.pop();
221                     }
```

```
222                     while(std::holds_alternative<Node*>(this->
                           current->content()) &&
223                              std::get<Node*>(this->current->content())
                                     != nullptr)
224                     {
225                         this->nodeStack.push(this->current);
226                         previousNodeCountStack.top() += 1;
227                         this->previousNodeCountStack.push(0);
228                         this->current = std::get<Node*>(current->
                               content());
229                     }
230                 }
231             }
232         };
233     };
234 }
235 #endif  // LAB2_TRIVARIATEHIERARCHICALLIST_H_
```

expression.h

```
1  #ifndef LAB2_EXPRESSION_H_
2  #define LAB2_EXPRESSION_H_
3
4  #include <iostream>
5  #include <vector>
6  #include <regex>
7
8  #include "trivariatehierarchicallist.h"
9
10 constexpr bool DEBUG = true;
11
12 namespace lab2
```

```
13 {
14     template<class T>
15     class Expression
16     : public lab2::TrivariateHierarchicalList<T, std::string, lab2
          ::OperationType>
17     {
18         typedef typename \
19             lab2::TrivariateHierarchicalList<T, std::string, lab2::
                OperationType>::Node \
20             ExpressionNode;
21
22     public:
23         Expression(const std::string &expression)
24         : lab2::TrivariateHierarchicalList<T, std::string, lab2::
             OperationType>()
25         {
26             auto expression_ = regex_replace(expression, std::regex
                ("^\\s*"), "");
27             expression_ = regex_replace(expression_, std::regex("\\
                s*$"), "");
28             expression_ = regex_replace(expression_, std::regex("\\
                s+"), " ");
29             if(DEBUG)
30             {
31                 std::cout << "Acquired string: \"" <<
32                      expression_ << "\"" << std::endl;
33                 std::cout << "Acquired string length: " <<
34                      expression_.length() << std::endl;
35             }
36             if(expression_.length() == 0 || regex_match(expression_
                ,
37                                                 std::regex(
                                                    "^\\s*$"
```

```
                                                                    )))
38              {
39                  this->parsingErrors.push_back("The provided
                       expression is empty");
40              }
41              else
42              {
43                  this->parse(this->head(), expression_.begin(),
                       expression_.end());
44              }
45          }
46
47          bool isCorrect()
48          {
49              std::stack<std::variant<T,bool>> executionStack;
50              executionErrors.resize(0);
51              this->checkNodes(executionStack);
52              if(executionStack.size() > 1)
53                  this->executionErrors.push_back(std::string("There
                       ") +
54                        (executionStack.size() > 2 ? "are " : "is ") +
55                        lab2::to_string(executionStack.size() - 1) +
56                        " unused operand" + (executionStack.size() > 2
                             ? "s " :
57                        " ") +  "left.");
58              if(this->parsingErrors.size() > 0 || this->
                    executionErrors.size() > 0)
59                  return false;
60              return true;
61          }
62
63          std::string getErrors()
64          {
```

```
65              if(this->parsingErrors.size() == 0 &&
66                 !(this->executionErrors.size() > 0 || this->
                      isCorrect()))
67                  return("There are no errors found.\n");
68
69              auto totalLength = 0;
70              for(auto error : this->parsingErrors)
71                  totalLength += error.length() + 1;
72              for(auto error : this->executionErrors)
73                  totalLength += error.length() + 1;
74
75              std::string result;
76              for(auto error : this->parsingErrors)
77                  result += error + "\n";
78              for(auto error : this->executionErrors)
79                  result += error + "\n";
80              return "Errors:\n" + result;
81          }
82
83      private:
84          std::vector<std::string> parsingErrors{};
85          std::vector<std::string> executionErrors{};
86
87          std::string::const_iterator parse(ExpressionNode *node,
88                                            std::string::
                                                 const_iterator current
                                                 ,
89                                            std::string::
                                                 const_iterator end)
90          {
91              static size_t depth = 0;
92              depth += 1;
93              if(DEBUG)
```

```
94              {
95                  std::cout << std::string(depth - 1, ' ') <<
96                       "|~~~~~~~~~~~" << std::endl;
97                  std::cout << std::string(depth - 1, ' ') <<
98                       "| the function \"parse\" was called" << std::
                            endl;
99                  std::cout << std::string(depth - 1, ' ') <<
100                      "| depth: " << depth << std::endl;
101             }
102
103             auto currentNode = node;
104
105             while(current != end && *current == ' ')
106             {
107                 current += 1;
108             }
109
110             if(current != end && *current == '(')
111             {
112                 if(depth == 1 && DEBUG)
113                     std::cout << std::string(depth - 1, ' ') <<
114                          "| the token \"(\" was acquired: " << std
                                ::endl;
115                 current += 1;
116                 while(current != end)
117                 {
118                     // skip spaces
119                     while(current != end && *current == ' ')
120                     {
121                         current += 1;
122                     }
123
124                     if(current == end)
```

```
125                 {
126                     this->parsingErrors.push_back("Expression
                           ended unexpectedly");
127                     return end;
128                 }
129 
130                 // get "("
131                 if(*current == '(')
132                 {
133                     if(DEBUG)
134                     {
135                         std::cout << std::string(depth - 1, ' '
                               ) <<
136                             "| the token \"(\" was acquired: "
                                 << std::endl;
137                     }
138                     currentNode->setContent(new ExpressionNode
                           ());
139                     current = parse(
140                         std::get<ExpressionNode*>(currentNode->
                               content()),
141                         current, end);
142                 }
143 
144                 // get ")"
145                 else if(*current == ')')
146                 {
147                     if(DEBUG)
148                     {
149                         std::cout << std::string(depth - 1, ' '
                               ) <<
150                             "| the token \")\" was acquired: "
                                 << std::endl;
```

```
151                     }
152                     if(std::holds_alternative<ExpressionNode*>(
                           currentNode->
153                        content())&& std::get<ExpressionNode*>(
                              currentNode->
154                        content()) == nullptr)
155                     {
156                         this->parsingErrors.push_back("The
                               provided"\
157                            "expression contains empty
                                  parentheses");
158                     }
159                     current += 1;
160                     break;
161                 }
162
163                 // get a variable
164                 else if(('A' <= *current && *current <= 'Z') ||
165                                             *current == '_' ||
166                         ('a' <= *current && *current <= 'z'))
167                 {
168                     std::smatch match;
169                     std::regex regex("^([_A-Za-z][_A-Za-z0-9]*)
                           ");
170                     std::regex_search (current, end, match,
                           regex);
171                     currentNode->setContent(match[1]);
172                     if(DEBUG)
173                     {
174                         std::cout << std::string(depth - 1, ' '
                               ) <<
175                            "| the token  \"" <<
```

```
176                         std::get<std::string>(currentNode
                               ->content()) <<
177                         "\" was acquired." << std::endl;
178                     }
179                     current += match[1].length();
180                 }
181
182                 // get a number
183                 // fractions of the pattern ".[0-9]+" are not
                       supported
184                 else if('0' <= *current && *current <= '9')
185                 {
186                     std::smatch match;
187                     std::regex regex("^([0-9]+(\\.[0-9]+)?)");
188                     std::regex_search (current, end, match,
                           regex);
189                     currentNode->setContent(lab2::stoT<T>(match
                           [1]));
190                     if(DEBUG)
191                     {
192                         std::cout << std::string(depth - 1, ' '
                               ) <<
193                             "| the token  \"" <<
194                             std::get<T>(currentNode->content()
                                ) <<
195                             "\" was acquired." << std::endl;
196                     }
197                     current += match[1].length();
198                 }
199
200                 // get an operator
201                 else if(*current == '+' || *current == '-' || *
                       current == '*' ||
```

```
202                     *current == '/')
203                 {
204                     if(*current == '+')
205                         currentNode->setContent(lab2::
                               OperationType::ADDITION);
206                     else if(*current == '-')
207                         currentNode->setContent(lab2::
                               OperationType::SUBTRACTION);
208                     else if(*current == '*')
209                         currentNode->setContent(lab2::
                               OperationType::MULTIPLICATION);
210                     else if(*current == '/')
211                         currentNode->setContent(lab2::
                               OperationType::DIVISION);
212                     current += 1;
213                     if(DEBUG)
214                     {
215                         std::cout << std::string(depth - 1, ' '
                               ) <<
216                             "| the token  \"" << lab2::
                                   to_string(std::get<\
217                             lab2::OperationType>(currentNode->
                                   content())) <<
218                             "\" was acquired." << std::endl;
219                     }
220                 }
221
222                 // ignore other chars
223                 else
224                 {
225                     this->parsingErrors.push_back("Unexpected
                           symbol: " +
```

```
226                                                     lab2::
                                                           to_string
                                                           (int(*
                                                           current))
                                                           );
227                         current += 1;
228                     }
229
230                     if(current != end && *current != ')')
231                     {
232                         currentNode->setNext(new ExpressionNode());
233                         currentNode = currentNode->next();
234                     }
235                 }
236             }
237             else
238             {
239                 this->parsingErrors.push_back("The symbol \"(\" is
                       abscent when necessary");
240             }
241             depth -= 1;
242             if(depth == 0)
243             {
244                 // report a presence of a character after the last
                       closing bracket
245                 if(current != end)
246                 {
247                     this->parsingErrors.push_back("The string
                           contains the "\
248                     "following character(s) after the last closing
                           bracket:\n\t\"" +
249                     std::string(current, end) + "\"");
250                 }
```

```
251             }
252             if(DEBUG)
253             {
254                 std::cout << std::string(depth, ' ') << "
                       |~~~~~~~~~~~~" << std::endl;
255                 if(depth > 0)
256                     std::cout << std::string(depth - 1, ' ') << "|
                           depth: " <<
257                               depth << std::endl;
258             }
259             return current;
260         }
261
262         // tries to compute the value of the stored expression in
               order to find
263         // all cases of division by 0
264         // "bool" represents an undecidable variable
265         void checkNodes(std::stack<std::variant<T,bool>>&
               executionStack)
266         {
267             size_t termCount = 1;
268             for(auto current = this->begin(), end = this->end();
269                 current != end; termCount += 1, ++current)
270             {
271                 // push a number onto the stack
272                 if(std::holds_alternative<T>(current->content()))
273                 {
274                     executionStack.push(std::get<T>(current->
                           content()));
275                     if(DEBUG)
276                     {
277                         std::cout << "~~~~~~" << std::endl;
```

```
278                     std::cout << "Term " + lab2::to_string(
                           termCount) +
279                       ", the value is "
280                       << std::get<T>(current->content())
281                       << std::endl;
282                 }
283             }
284             // push an unknown value onto the stack
285             else if(std::holds_alternative<std::string>(current
                   ->content()))
286             {
287                 executionStack.push(true);
288                 if(DEBUG)
289                 {
290                     std::cout << "~~~~~~" << std::endl;
291                     std::cout << "Term " + lab2::to_string(
                           termCount) +
292                       ", the value is undecidable" << std::
                             endl;
293                 }
294             }
295             // execute an operation
296             else if(std::holds_alternative<lab2::OperationType
                   >(current->content()))
297             {
298                 auto previousNodeCount = current.
                       getPreviousNodeCount();
299                 auto operationType = std::get<lab2::
                       OperationType>(current->content());
300                 if(DEBUG)
301                 {
302                     std::cout << "~~~~~~" << std::endl;
```

```
303                 std::cout << "Term " + lab2::to_string(
                        termCount) +
304                         ", the operation is \"" <<
305                         lab2::to_string(operationType) + "\""
306                         << std::endl;
307             }
308             // if the operation is called as niladic,
                    report an error
309             if(previousNodeCount == 0)
310             {
311                 this->executionErrors.push_back("Term "
312                         + lab2::to_string(termCount) + ": The
                            operation \"" +
313                         lab2::to_string(operationType) +
314                         "\" has too few (0) arguments.");
315             }
316             // if the operation is called as ternary or
                    ..., report an error
317             else if(previousNodeCount > 2)
318             {
319                 this->executionErrors.push_back("Term "
320                         + lab2::to_string(termCount) + ": The
                            operation \"" +
321                         lab2::to_string (operationType) + "\"
                            has too many (" +
322                         lab2::to_string(previousNodeCount) + "
                            ) arguments.");
323                 // remove all its arguments from the stack
324                 while(previousNodeCount --> 0)
325                 {
326                     if(DEBUG)
327                     {
```

```
328                         std::cout << "Term " + lab2::
                               to_string(termCount) +
329                             ", the operand " +
330                             lab2::to_string(
                                   previousNodeCount + 1) +
331                             " is removed from the stack"
                                   << std::endl;
332                     }
333                     executionStack.pop();
334                 }
335                 // push an unknown value onto the stack as
                       its result
336                 executionStack.push(true);
337                 if(DEBUG)
338                 {
339                     std::cout << "Term " + lab2::to_string(
                           termCount) +
340                         ", the result is meaningless" <<
                               std::endl;
341                 }
342             }
343             // if the operation is called as unary, ...
344             else if(previousNodeCount == 1)
345             {
346                 // if it is "-", try to execute it
347                 if(operationType == lab2::OperationType::
                       SUBTRACTION)
348                 {
349                     if(std::holds_alternative<T>(
                           executionStack.top()))
350                     {
351                         auto operand = std::get<T>(
                               executionStack.top());
```

```
352                     if(DEBUG)
353                     {
354                         std::cout << "Term "
355                                 + lab2::to_string(
                                    termCount) +
356                                 ", the operand is " <<
                                    operand
357                                 << std::endl;
358                     }
359                     executionStack.pop();
360                     executionStack.push(-operand);
361                 }
362                 else
363                 {
364                     if(DEBUG)
365                     {
366                         std::cout << "Term "
367                                 + lab2::to_string(
                                    termCount) +
368                                 ", the operand is
                                    undecidable"
369                                 << std::endl;
370                     }
371                     executionStack.pop();
372                     executionStack.push(true);
373                 }
374             }
375             // if it is not "-", report an error
376             else
377             {
378                 this->executionErrors.push_back("Term "
379                       + lab2::to_string(termCount) +
380                       ": The operation \"" +
```

```
                        lab2::to_string(operationType) +
                        "\" has too few (1) arguments.");
                executionStack.pop();
                if(DEBUG)
                {
                    std::cout << "Term " + lab2::
                       to_string(termCount) +
                         ", the operand " +
                         lab2::to_string(
                            previousNodeCount + 1) +
                        " is removed from the stack"
                            << std::endl;
                }
                executionStack.push(true);
                if(DEBUG)
                {
                    std::cout << "Term " + lab2::
                       to_string(termCount) +
                         ", the result is meaningless"
                             << std::endl;
                }
            }
        }
        // if the operation is called as binary, ...
        else if(previousNodeCount == 2)
        {
            // retrieve the operands in reverse order
            std::variant<T,bool> operand2 =
               executionStack.top();
            executionStack.pop();
            std::variant<T,bool> operand1 =
               executionStack.top();
            executionStack.pop();
```

```
407                 if(DEBUG)
408                 {
409                     std::cout << "Term " + lab2::to_string(
                           termCount) +
410                           ", the operand 1 is " + (
411                           std::holds_alternative<bool>(
                              operand1) ?
412                           "undecidable" :
413                           lab2::to_string(std::get<T>(
                              operand1)))
414                           << std::endl;
415                     std::cout << "Term " + lab2::to_string(
                           termCount) +
416                           ", the operand 2 is " + (
417                           std::holds_alternative<bool>(
                              operand2) ?
418                           "undecidable" :
419                           lab2::to_string(std::get<T>(
                              operand2)))
420                           << std::endl;
421                 }
422                 // check for division by 0
423                 if(std::holds_alternative<T>(operand2) &&
424                    operationType == lab2::OperationType::
                         DIVISION &&
425                    std::get<T>(operand2) == 0)
426                 {
427                     this->executionErrors.push_back("Term "
428                           + lab2::to_string(termCount) +
429                           ": Division by 0 encountered.");
430                     executionStack.push(true);
431                     if(DEBUG)
432                     {
```

```
433                 std::cout << "Term " + lab2::
                        to_string(termCount) +
434                       ", the result is meaningless"
                             << std::endl;
435             }
436         }
437         // check for undecidable operands
438         else if(std::holds_alternative<bool>(
                operand1) ||
439                 std::holds_alternative<bool>(
                        operand2))
440         {
441             executionStack.push(true);
442             if(DEBUG)
443             {
444                 std::cout << "Term " + lab2::
                        to_string(termCount) +
445                       ", the result is undecidable"
                             << std::endl;
446             }
447         }
448         // if neither division by 0 nor undecidable
                operands were
449         // encountered, execute the operation and
                save its result
450         else
451         {
452             T result;
453             if(operationType == lab2::OperationType
                    ::ADDITION)
454             {
455                 result = std::get<T>(operand1) +
456                          std::get<T>(operand2);
```

```
457                 }
458                 if(operationType == lab2::OperationType
                       ::SUBTRACTION)
459                 {
460                     result = std::get<T>(operand1) -
461                              std::get<T>(operand2);
462                 }
463                 if(operationType == lab2::OperationType
                       ::MULTIPLICATION)
464                 {
465                     result = std::get<T>(operand1) *
466                              std::get<T>(operand2);
467                 }
468                 if(operationType == lab2::OperationType
                       ::DIVISION)
469                 {
470                     result = std::get<T>(operand1) /
471                              std::get<T>(operand2);
472                 }
473                 executionStack.push(result);
474                 if(DEBUG)
475                 {
476                     std::cout << "Term " + lab2::
                           to_string(termCount) +
477                               ", the result is " +
478                               lab2::to_string(result) << std
                                  ::endl;
479                 }
480             }
481         }
482         // ignore operands that oppear after an
               operator
483         if(current->next() != nullptr)
```

```
484                         {
485                             while(current->next() != nullptr)
486                             {
487                                 termCount += 1;
488                                 ++current;
489                                 this->executionErrors.push_back("Term "
                                        +
490                                      lab2::to_string(termCount) +
491                                      " comes after an operator, and
                                           thus is ignored");
492                                 if(DEBUG)
493                                 {
494                                     std::cout << "~~~~~~~" << std::endl;
495                                     std::cout << "Term " + lab2::
                                           to_string(termCount) +
496                                             ", the value is ignored" <<
                                                std::endl;
497                                 }
498                             }
499                         }
500                     }
501                 }
502                 if(DEBUG)
503                     std::cout << "~~~~~~~" << std::endl;
504             }
505         };
506 }
507
508 #endif  // LAB2_EXPRESSION_H_
```

main.cpp

```cpp
#include <iostream>
#include <string>

#include "expression.h"

using std::cin;
using std::cout;
using std::endl;

using lab2::Expression;

int main(int argc, char* argv[])
{
    if(argc > 2)
    {
        cout << "Too many command line arguments were provided." <<
            endl;
    }

    std::string line;
    if(argc == 2)
    {
            line += std::string(argv[1]);
    }
    else
    {
        cout << "Enter an expression:" << endl;
        std::getline(cin, line);
    }

    Expression<double> expression(line);
    cout << expression.represent() << endl;
    if(expression.isCorrect())
```

```
33      {
34          cout << "The given expression \x1b[1mis\x1b[0m correct." <<
                endl;
35      }
36      else
37      {
38          cout << "The given expression is \x1b[1mnot\x1b[0m correct.
               " << endl;
39          cout << expression.getErrors();
40      }
41      cout << endl; // so consecutive calls would be distinct
42  }
```